# MEMCOM-MATLAB-API

## Version 1.0.0

M.J.J. Nijhof

**U**niversity of **T**wente
**SMR** *Engineering & Development*

April 4, 2005

# Contents


**1 Introduction** **4**
- 1.1 General introduction . . . 4
- 1.2 Acoustics . . . 6
- 1.3 Outline . . . 6

**2 Task description** **8**
- 2.1 Terminology and software packages . . . 8
  - 2.1.1 Terminology . . . 8
  - 2.1.2 Software Packages . . . 10
- 2.2 Main task and subtasks . . . 10
  - 2.2.1 Description of the work . . . 10
  - 2.2.2 Assessment of subtasks . . . 11
- 2.3 Course for the remainder of the report . . . 11

**I A coupling between MEMCOM and MATLAB** **12**

**3 The need for a MEMCOM - MATLAB coupling** **13**
- 3.1 Task description . . . 13
- 3.2 Requirements and requests . . . 14

**4 Connecting to MEMCOM and MATLAB** **16**
- 4.1 Connecting to MATLAB . . . 16
  - 4.1.1 MATLAB - engine and APIs . . . 16
  - 4.1.2 MATLAB mex-functions . . . 17
  - 4.1.3 Executing shell commands in MATLAB . . . 18
- 4.2 Connecting to MEMCOM . . . 18
- 4.3 Choosing a method . . . 19
  - 4.3.1 Creating a coupling with the MATLAB - API . . . 19
  - 4.3.2 Creating a coupling using mex-functions . . . 19
  - 4.3.3 Choosing mex-functions . . . 20

4.4 Choosing a language . . . . . 20
4.4.1 Fortran . . . . . 20
4.4.2 C . . . . . 21
4.4.3 choosing C . . . . . 21

**5 Program features and data structures 22**
5.1 Features of the MEMCOM - C-API . . . . . 22
5.1.1 How MEMCOM works . . . . . 22
5.1.2 Data structures and supported data types . . . . . 23
5.2 Features of MATLAB and mex-functions . . . . . 24
5.2.1 important MATLAB features . . . . . 24
5.2.2 How mex-functions work . . . . . 25
5.2.3 Data structures and supported data types . . . . . 25
5.2.4 name conventions . . . . . 26
5.3 Features of B2000 . . . . . 27
5.3.1 Data structures . . . . . 27
5.3.2 Naming conventions . . . . . 28

**6 Defining the mex-functions 30**
6.1 General design decisions . . . . . 30
6.1.1 Writing a function library . . . . . 31
6.1.2 Error handling . . . . . 31
6.2 Implementation of the mex-functions . . . . . 31
6.2.1 Assessment of the required mex-functions . . . . . 31
6.2.2 General structure of a mex-function . . . . . 32

**7 Evaluation and recommendations 34**
7.1 Evaluation . . . . . 34
7.2 Recommendations . . . . . 36

**II An iterative eigenvalue solver in MATLAB 38**

**8 Viscous elements in B2000 39**
8.1 Introduction to viscous elements 95 and 96 . . . . . 39
8.2 Task description . . . . . 40
8.2.1 Subtasks . . . . . 40
8.2.2 Requirements and requests . . . . . 40

**9 Assembler for asymmetric complex matrices 42**
9.1 Design decisions and boundary conditions . . . 42
9.1.1 Data structures . . . 42
9.1.2 Complex numbers . . . 43
9.1.3 Name conventions . . . 43
9.1.4 Choosing a language . . . 44
9.2 Implementation . . . 45
9.2.1 The main function . . . 45
9.2.2 The control module . . . 46
9.3 B2COMPARE . . . 47
9.3.1 B2COMPARE as an utility function . . . 47
9.3.2 The working of B2COMPARE . . . 48

**10 Results, evaluation and recommendations 49**
10.1 Results . . . 49
10.1.1 Element and system matrices . . . 49
10.1.2 Converged eigenvalues . . . 50
10.2 Evaluation . . . 51
10.2.1 Requirements and requests . . . 51
10.2.2 Evaluation of the results . . . 52
10.3 Recommendations . . . 52

**11 An iterative solver in MATLAB 53**
11.1 Layout of the solver . . . 53
11.1.1 Function blocks . . . 53
11.1.2 Iterative process . . . 53
11.2 Working of the solver . . . 54
11.2.1 Option structure . . . 54

**Bibliography 57**

**A Defining the mex-functions 58**
A.1 Opening and closing MEMCOM . . . 58
A.2 Data transfer to and from MEMCOM . . . 60
A.3 Database management . . . 64
A.4 B2MATLIB function library . . . 67

**B B2MATLAB 70**
B.1 Working . . . 70
B.2 Processor layout . . . 71

# 1

# Introduction

This report is written as a conclusion of a three month trainee period at SMR (Switzerland). The internship is a part of the five year curriculum of the education of a mechanical engineer at the faculty of engineering technology at the University of Twente. It is to be carried out in the final year before working on the master thesis.

The assignment consisted of developing a new feature of the B2000 finite element environment. Since SMR is the initiator of the B2000 software and is responsible for its maintenance, it was sensible to carry out the assignment at the SMR office in Switzerland.

## 1.1 General introduction

**The concept of B2000** B2000 is not only a computer program or a finite element environment, it is a concept. In a conventional software development cycle it can take an extensive period of time before the requirements and request of the users are translated to usable code.

The concept behind B2000 is that every party can contribute to the program by adding its own code or improving the code and routines of others. The principal idea is that the development of new features and improvement of existing features is much faster when the users implement the improvements themselves. This concept makes B2000 very attractive for institutions who are involved in research and development of new computational techniques.

On the other hand, this concept requires a central supervision instance to ensure the consistency of the developments and to maintain quality of the produced code. Since every party has its own requirements, request, programming techniques and vision, the code is bound to become unorganized, unreadable and eventually unusable without supervision.

This is where the importance of SMR, the initiator of the B2000 environment lies. Since the first release, SMR has been maintaining B2000, assuring the quality of the code. SMR makes sure that changes and new features do not cause complications with the working of the existing code and it offers facilities that endorse structure and consistency of the code.

**Maintenance of the code** The different parties involved in developing B2000 are (welcomed but) not obliged to share the improvements and additions they make to the program. In some cases a party is not allowed or willing to share its work with the rest of the B2000 community. It is impossible for SMR to guarantee that parts of code which are withheld will continue to work without complications as the rest of the B2000 code evolves. If, in such a case, the code is not maintained by its creators, it will ultimately become unusable in newer versions of B2000.

The problem described above ultimately led to this assignment. Between 1993 and 1995 a number of acoustic elements were developed for the B2000 environment at the department of applied mechanics and plastics of the University of Twente. The implemented elements required an eigenvalue solver for asymmetric complex matrices which was already available in B2000 and was developed by the NLR. The code of the solver was not shared with the rest of the B2000 community and as such was not maintained or updated to work with the new B2000 release. The NLR allowed the University of Twente to use a binary version[1] of the solver.

**A coupling between MEMCOM and MATLAB** Recently the department of applied mechanics was involved in research concerning these acoustic elements and the need arose to be able to work with them in newer versions of B2000. The code of the corresponding eigenvalue solver, however, is at present not available to the department. A (temporary) solution for this problem is to utilize the eigenvalue solvers of MATLAB that are suited to work with asymmetric complex matrices. This solution requires the transfer of data created by B2000 to the MATLAB environment and vice versa. Such a coupling between B2000 and MATLAB was not yet available.

The usefulness of such a coupling is not limited to this specific problem solely. With the coupling, a vast number of mathematical routines available in MATLAB, become available for the B2000 programmers without having

[1] A binary version of a computer program can not be altered. No changes or improvements can be made to such code.

to implement them themselves. The possibility to use MATLAB from within B2000 or vice versa facilitates rapid implementation and testing of new algorithms. It also avoids having to implement complex algorithms in the first stage of the design of new B2000 routines. On top of that, it can be used for educational means. Most students have experience in programming with MATLAB, but not with other programming languages. Thus, a coupling between B2000 and MATLAB can help to demonstrate the working of B2000 to students, which are potential new B2000 users, in an easy and accessible way.

## 1.2 Acoustics

**The viscothermal effect** The theory behind the acoustic elements that are to be integrated in the B2000 environment is explained in the thesis of Marco Beltman [1]. He describes the phenomenon of acoustics by commenting on the following quote of the Webster's revised unabridged dictionary: "acoustics: the science of sounds, teaching their nature, phenomena and laws". He adds to this quote that "Sound is generated by motion of particles. It is a compression and refraction of the medium." Beltman's thesis deals with pressure waves in a gas trapped in thin layers or narrow tubes. In these cases viscous and thermal effects can have significant effects on the propagation of the sound waves.

**Element code** A considerable part of the thesis is concerned with numerical calculations involving models of viscothermal wave propagation. Chapter five of Beltman's thesis describes the development, implementation and analytical and experimental validation of the finite element model based on the low reduced frequency equation. The element code available of these so-called viscothermal elements is the result of this research and was initially programmed by Marco Beltman. Frank Grooteman adapted the element code such that it can be used in B2000 and adheres to the B2000 standards.

## 1.3 Outline

This report consists of two parts. The first part deals with the design and the implementation of a coupling between MEMCOM and MATLAB and a new B2000 processor called B2MATLAB. The second part is concerned

with an application of this coupling. It describes an iterative routine in MATLAB for solving the coupled eigenfrequencies and modes of a structure that consists of viscothermal and structural elements.

In chapter 2 the task description is being examined and the decision to split the assignment up in two independent problems is explained. Furthermore, a course for the remainder of this report is plotted.

Chapter 3 deals with the specific needs and requests to which the coupling between MEMCOM and MATLAB is subjected.

The next chapter deals with the question how a coupling might best be constructed without going into detail. In addition, it addresses a number of generic design questions that apply to software design in general.

In Chapter 5 the features and data structures of the MEMCOM - C-API, the MATLAB mex-functions and B2000 are explored.

The next chapter describes how the the different task are grouped together to form functions which the user can call from within MATLAB. It also deals with the design and implementation of the mex-functions an the function library that is designed.

Chapter 7 consist of a short summary and an evaluation of the work presented in this part. The chapter ends with a number of recommendations for further development of the MEMCOM - MATLAB coupling.

Chapter 8 describes the second part of the assignment in more detail, and it gives a global insight in the process of a B2000 finite element calculation.

The requirements of an iterative eigenvalue solver are stated in chapter 8.

In chapter 9 the design and implementation of an asymmetric complex system matrix assembler is described.

Chapter 11 deals with the implementation of the iterative solver in MATLAB and the use of the MEMCOM - MATLAB coupling.

An evaluation of the design of the assembler and iterative solver and the results of a number of test sums is presented in chapter 10.

# 2

# Task description

The first section of this chapter is used to comment on some of the terminology and software packages that are frequently used in this report. Thereafter, the task that was to be carried out is examined and a course for the remainder of this report is plotted.

## 2.1 Terminology and software packages

### 2.1.1 Terminology

The concepts presented in the next four paragraphs are involved in a number of important design decisions throughout the report. The descriptions are not presumed to be complete, but rather to function as a short introduction to the terminology used.

**API** The API[1] of a computer program is an interface which allows communication with other (by the user written) programs. The interface is in fact a set of functions or subroutines that must be included in the new program and can then be called from within it. APIs are language specific, that means that for each language a program wants to communicate in, a separate API must be developed.

**Function overloading** This terminology is used when a number of functions which have different implementation, but most often similar properties, share the same name. The interpreter or compiler of the computer

[1] API is the abbreviation of Application Program Interface.

language in which the function is written, determines which implementation is to be used. This feature is often used when for instance two functions perform the same operation, but the input and output types of the functions differ from one another.

Another good reason to use function overloading, is when one of the input arguments of a function is omitted and a default setting is used. Two functions perform in that case the same operations on the input data, but the number of inputs (or outputs) may differ.

**Data structure** A data structure is the way in which the data a program operates on is organized. One of the first things a programmer has to decide when writing a program is how to organize the data that is to be processed. When writing an addition to an existing program or working with output data of another program it is necessary to study the data structure used and to find a way to make your own data structure meet that of the other program.

**OO or OOP** Object oriented programming (OOP), is a programming technique that makes use of objects. These objects (often referred to as classes) are a feature supported in some computer languages[2] that helps organize the data of a computer program and the functions that operate on this data. The data structures (or variables) and corresponding functions are grouped together to form objects. This representation can help a programmer to think of data and functions in a way we think of every day objects, like a car, a house or a tree. The following simple example illustrates the concept:

The object car can have for instance the properties color and license plate number. A function could be implemented that only returns the number of the license plate if the car has a certain specified color. This function belongs to the object car and as such, can not be called to operate on other objects, which is quite sensible. After all, it would not make much sense to allow this function to operate on for instance an object describing a tree, since a tree does not have a license plate number.

[2]The languages discussed in this report that support object oriented programming are C++ and Python.

### 2.1.2 Software Packages

The following three paragraphs are comprised of short introductions to the three software packages involved in this assignment. A more detailed description of their working and internal structure is presented in chapter 5.

**MEMCOM** The MEMCOM system, developed by SMR, is a data management system for scientific engineering applications. It allows for quick and straight forward data storage and retrieval. The package is written in C and comes with an extensive C-, Fortran-, and Python-API.

**B2000** The B2000 finite element environment was developed by SMR. The users are given access to the source code and are free to make their on adjustments or improvements to it. B2000 consists of a number of independent processors which can be run separately. The separate processors can also be combined to form a macro processor. The backbone of the B2000 package is the MEMCOM data manager. Each processor reads its data from a MEMCOM database and writes its output back to the same (or another) MEMCOM database for further processing.

**MATLAB** The MATLAB suite is a well known math-package developed by THE MATHWORKS used by scientists the world over. The package comes with a vast variety of solvers and allows vector and matrix manipulations. The command line interpreter and scripting language facilitate quick and easy programming of scripts containing internal MATLAB commands or user specified commands.

## 2.2 Main task and subtasks

### 2.2.1 Description of the work

The main task of this work consists of integrating viscous elements (number 95 and 96) and their coupling elements (number 104 and 105) in B2000 version 2.5. A MATLAB eigenvalue solver which calculates the lowest eigenfrequencies of a system is to be integrated in the B2000 environment. The possibilities to create a more general coupling between MEMCOM and MATLAB are explored.

### 2.2.2 Assessment of subtasks

It is evident that the main task consists of two main subtasks and that the order in which these subtask are to be carried out is fixed.

**First subtask** Create a coupling between MEMCOM and MATLAB. The coupling should allow MATLAB to use information stored on a MEMCOM database in its calculations and write the results of these calculations back to the MEMCOM database.

**Second subtask** Integrate the elements mentioned above in B2000, solve the eigenvalue problem in MATLAB, export the results back to a MEMCOM database and compare them with the results obtained with the old B2000 solver.

## 2.3 Course for the remainder of the report

The two subtasks can be carried out subsequently. There is no need to perform the design process of the MEMCOM -MATLAB coupling parallel to the integration of the viscous elements in B2000. Therefore the report is split up into two individual parts. The first part will cover the development of the MEMCOM -MATLAB coupling and the second part will deal with the integration of the viscous elements in B2000.

# Part I

# A coupling between MEMCOM and MATLAB

# 3

# The need for a MEMCOM - MATLAB coupling

As mentioned in the introduction, the code of the acoustic elements developed by Marco Beltman was no longer operational in the newer version of B2000 and the code of the corresponding eigenvalue solver is not available at present. A (temporary) solution for the second problem is utilizing the eigenvalue solvers of MATLAB. This solution requires the transfer of data outputted by B2000 to the MATLAB environment and data outputted by MATLAB back to the B2000 environment.

In the next section the task description of the first subtask is examined. Afterward, the specific needs and requests to which the MEMCOM - MATLAB coupling is subjected are being explored.

## 3.1 Task description

The description of this subtask as given in the previous chapter is not sufficiently precise. It describes a coupling between MEMCOM and MATLAB without taking into account that the coupling should act as a first between MATLAB and a MEMCOM database created with B2000. It is best to keep the data structures of the B2000 system in mind when designing the coupling (Since the first application of the coupling is intended to be used with B2000). However, the coupling must not be too type specific to this case, because it might be used for other (yet unforeseen) purposes in the future. The above-mentioned, gives rise to the requirements and request listed in the next section.

## 3.2 Requirements and requests

### Requirements

The MEMCOM - MATLAB coupling must account for the following operations:

- Read and write vectors or full two dimensional matrices, stored on a MEMCOM database, from and too MATLAB while maintaining the size and dimension.
- Read and write descriptor tables, stored on a MEMCOM database, from and to MATLAB
- Read and write relational tables, stored on a MEMCOM database, from and to MATLAB
- Vectors and Matrices must be of type `double` (64 bit) or `int` (32 bit)
- Information in descriptor tables must be of type `double`, `int` or `char`

The coupling between MATLAB and a B2000 - MEMCOM database requires an extra demand:

- Read and write two dimensional matrices in skyline- or compressed row storage format, stored on a MEMCOM database, from and to MATLAB

### Requests

The MEMCOM - MATLAB coupling is subjected to the requests stated below. A brief explanation is given in of why the requests are in fact request and not requirements.

- Read and write more dimensional matrices (up to five, the limit of MEMCOM ) from and to MATLAB.
- Read and write vectors and matrices of other types than `double` or `int` (MEMCOM supports among others `real`, `char` and `unknown`)
- Expand the number of types that are allowed in descriptor tables and relational tables to the same number of types that MEMCOM allows.

The first request is not a first priority, since MATLAB can handle more dimensional matrices but does not support it as thoroughly as two dimensional matrices. In fact MATLAB is designed to work with two dimensional matrices. (For instance scalars are represented internally as one by one matrices and the number of functions available for more dimensional matrices is limited.)

The nature of MATLAB makes that the second and third request are also not first priority. All matrices in MATLAB are standard of type `double`. It is possible to force MATLAB to use another data type for vectors and matrices, but these vectors and matrices can than only be used for storage. In other words, almost none of the operations or functions support these data types. A solution around this problem is to convert all numerical data to type `double` within MATLAB, but it hardly makes sense to convert data to type `double`, perform operations on it and convert it back to its original type before writing it back to MEMCOM. It is more elegant to make sure that the other program that reads and writes from and to the MEMCOM database (for instance a B2000 processor) uses the data types supported in MATLAB.

# 4

# Connecting to MEMCOM and MATLAB

This chapter deals with the question how a coupling might best be constructed without concerning with the details. In addition it addresses a number of generic design questions pertaining to software design in general which are relevant to this case.

## 4.1 Connecting to MATLAB

There are two methods to connect a program to MATLAB. You can call the MATLAB - engine from within your own program or call your programs and functions from within a MATLAB session. Both methods and their advantages and disadvantages are discussed below.

### 4.1.1 MATLAB - engine and APIs

The MATLAB package offers the possibility to make use of the MATLAB - engine from within external programs. The package comes with an extensive C/C++, Fortran and Java API. The APIs consist of a number of functions that let you access almost all features available in MATLAB. It is possible to write a piece of code that at a certain point starts a MATLAB - engine session. It is then possible to feed data to the engine, perform operations on it and extract the results from the MATLAB workspace.

**Advantages** An advantage of this method is that all programming can be done in the same language. The whole program including MATLAB calls can be coded in the same environment and consists of a single file.

This offers simplicity, in the sense that calls to the different processors (which may or may not use MATLAB routines) can be made from one macro processor. There is no need to start up a MATLAB session and type in commands or run scripts in it.

**Disadvantages** A disadvantage of this method, is that it requires a good amount of knowledge of the MATLAB - APIs on the part of the user. Anyone who ever wants to use such a coupling must not only be well acquainted with the normal MATLAB commands but also with its equivalents in the API that is used, This is a result of the fact that all MATLAB calls will be hard coded, e.g. all commands will be explicitly included in the code of the program.

A way to resolve this, is to call MATLAB scripts instead of single commands, as their contents may vary while their calls remain the same. The MATLAB engine, however, will stop executing a script when an error occurs. This makes debugging very difficult and time consuming.

Another big drawback of this approach is that the feedback on errors and results is in general indirect. A number of bugs will only reveal themselves during runtime and not during compile time.

### 4.1.2 MATLAB mex-functions

Another way to utilize MATLAB in combination with code written by the user, consists of using MATLAB mex-functions. This approach is in fact the opposite of the use of the MATLAB - APIs. A MATLAB session is launched, after which the command line or MATLAB scripts are used to perform all operations.

Mex-functions are functions written by the user in either Fortran or C (not C++). The functions are compiled from the MATLAB command line and can be called afterwards just as one would call an internal MATLAB command. The function always receives a pointer to an array of incoming MATLAB data structures and a pointer to an array of outgoing MATLAB data structures. There are even more functions available to operate on these data structures than in the MATLAB - API. It is also possible to create, destroy or manipulate other data structures in the workspace.

In addition, one can use function overloading, write one's own error messages and exceptions, and extend to the internal MATLAB help system to make debugging of programs which use these mex-functions easier.

**Advantages** There appears to be one big drawback to this approach, but the solution to the underlying problem turns it into its biggest advantage. When using this method, it is necessary to start up a MATLAB session parallel to any other program that is to operate on the data. (Note that the programs must act on the data successively not simultaneously). This causes complications, because each time one would like to run a batch program that involves a MATLAB operation, the MATLAB scripts and other programs have to be started manually. This problem can be resolved with full satisfaction by making use of MATLAB's command line possibilities. (In the next section a description is given of the methods to executed shell commands from the MATLAB prompt.) In this way the MATLAB prompt functions as a normal shell prompt. All operations and calls to other programs can be made from the MATLAB command line and even from one single MATLAB script. This makes it possible to operate B2000 and MATLAB from a single software environment.

**Disadvantages** A disadvantage of this method is, that the choice of languages is limited to C or Fortran. (C++ can not be used because MATLAB does not have an internal C++ compiler) This means that it is not possible to make use of OO programming techniques or function overloading when designing the coupling.

### 4.1.3 Executing shell commands in MATLAB

MATLAB facilitates command line execution of shell commands and -scripts in two ways. The simplest method is to begin a command with an exclamation point. All data on that line will be interpreted by MATLAB as shell commands. A second method is to use the command 'unix()'. This function takes a string containing the shell commands and returns the status and result if necessary. Both methods have the same effect, but the second method allows for a variable string as input for the command string to be executed.

## 4.2 Connecting to MEMCOM

MEMCOM has an elaborate interface to three prominent computer languages, C/C++, Fortran and Python. All MEMCOM facilities are accessible through these APIs. The utilization of most functions is very similar for the

C- and Fortran-API. (The Python variant, with Python being an OO programming language, has a completely different interface.) The functions available are divided into a number of groups. In the design of the MEMCOM - MATLAB coupling, functions of two groups are used (namely the database manager functions and relational table manager functions).

The nature of the MEMCOM - API has little influence on the two design questions at hand; "How to connect MEMCOM with MATLAB ?" and "What language should be used?", because there is only one way to connect to MEMCOM and the API-functions of importance do not differ that much for the C- and Fortran-API. (The use of the Python language is not an option, since it it can not be used in combination with the Matlab-APIs and mex-functions.)

## 4.3 Choosing a method

Both methods that are described above offer the possibility to run scripts, batches or macro processors while taking advantage of the mathematical routines available in MATLAB . The biggest difference between the two is the skill required to use the coupling.

### 4.3.1 Creating a coupling with the MATLAB - API

When the MATLAB - API and -engine is used, the programmer who intents to use the coupling must have a profound knowledge of C/C++ or Fortran, MATLAB and the MATLAB - API functions. In a way, with each program that is written, the MEMCOM - MATLAB coupling is written all over again. This is the result of the need to compile a program in C/C++ or in Fortran. All MATLAB commands will thus be hard coded. Furthermore, the programs devised with this method will be very hard to debug, because a number of errors will only be noticed during runtime and not during compile time.

### 4.3.2 Creating a coupling using mex-functions

There are less programming skills required when mex-functions (as described in 4.1.2) are used to device the coupling. In a way, this method can be seen as a new MEMCOM - API, a MEMCOM - API for MATLAB . The coupling has to be programed only once and the mex functions can stay unaltered. The user of the coupling only needs to know how to use these functions and how to program in MATLAB . Programming experience

in C/C++, Fortran or of the MEMCOM - APIs is not required. Thus, this coupling can also be used by relatively inexperienced programmers. The possibilities to write error messages and exceptions and append to the internal MATLAB help system, makes debugging of new scripts relatively easy.

### 4.3.3 Choosing mex-functions

Based on the considerations mentioned above, it may be quite clear that the choice is made to design the coupling inside MATLAB with the help of mex-functions. The skills required on the user are far smaller when making use of mex-functions. Hence, the coupling can also be used by less experienced programmers and this is most likely the group who will benefit from the coupling. The coupling will be used mainly for a fast check on the efficiency and accuracy of newly developed methods. To perform a fast check, it is most convenient to write the entire routine in MATLAB and not have to worry about things like for instance variable declaration and data structures that come with a lower level language like C.

To offer B2000 users the possibility to call the MATLAB engine (including the mex-functions of the MEMCOM - MATLAB coupling that is to be developed) from within their own routines, a new B2000 processor called B2MATLAB was developed. The design and working of this processor is commented on in Appendix B.

## 4.4 Choosing a language

The choice to make use of mex-functions narrows the number of languages, in which the coupling can be implemented, by one. MEMCOM allows the use of C/C++, Fortran and Python. MATLAB mex-functions can be written in either C or Fortran, so C++ is no longer an option. The two remaining options are reviewed shortly below. In section 4.4.3 it is explained why C is the language of choice in this case.

### 4.4.1 Fortran

B2000 is written mainly in Fortran 77. Even at this point in time, a part of the new code written for B2000 is written in Fortran. This is however, in no way a reason to choose Fortran as a language to device a coupling

between B2000 and MATLAB[1]. Fortran 77 has, unlike C, a number of limitations that make it relatively unattractive to use. The biggest limitations are the lack of dynamic memory and the absence of data structures. The size of an array of elements must be known at compile time and cannot be determined or altered at runtime. This yields severe limitations in its use in a MEMCOM - MATLAB coupling, since the size of the data structure which is to be parsed from one program to the other is never known in advance. A way to work around this is to make use of the dynamic memory functions (DMM) available in the MEMCOM - Fortran-API. The drawbacks of DMM are inherent to the Fortran language. It is not always possible to control the access to illegal memory cells during run-time, i.e. the application program may overwrite portions of memory or worse, portions of code (see [5]). Another limitation is the lack of pointers. The MATLAB - Fortran-API resolves this problem mostly by copying data to a Fortran array (instead of parsing a pointer as is the case in the C-API). This results in the case of the MEMCOM - MATLAB coupling in unnecessary copying of vast amounts of data.

### 4.4.2 C

Both MEMCOM and MATLAB offer excellent means of connecting to them from programs written in C. The language offers a number of build-in features that can be used most effectively in designing a MEMCOM - MATLAB coupling. The use of dynamic memory and pointers result in clean readable code that can be easily maintained and extended.

### 4.4.3 choosing C

It should be quite clear that the decision to develop the MEMCOM - MATLAB coupling in the language C is an easy one. Both dynamic memory and pointers are build-in features of C. To accomplish the same facilities in Fortran one has to work around the limitations of the language and use it in a way it was never intended to be used. This results in somewhat 'unnatural' programming. And what is more, both the development of MEMCOM and MATLAB are more focused on C written applications than on applications written in Fortran. So, it is clear that C is the language to be used for this application.

[1] All data transfer between B2000 and MATLAB goes through MEMCOM databases and MEMCOM is written completely in C.

# 5

# Program features and data structures

In this chapter a number of the features and the data structures of the MEMCOM - C-API, the MATLAB mex-functions and B2000 are explored.

## 5.1 Features of the MEMCOM - C-API

The MEMCOM - C-API consist of four groups of functions. In order to design the MEMCOM - MATLAB coupling only the functions in the following two groups are of interest: Database manager functions and relational table manager functions. The utility functions and client/server functions are of no importance. Separation of the database manager functions and the relational table manager functions is mainly due to the difference in data structure between ordinary datasets and relational tables. Beside all functions that operate on datasets, the group of database manager functions also contains the functions to open, close and inquire after the properties of a database. The next two subsections consist of a summary of the introduction of the MEMCOM Reference Manual [5].

### 5.1.1 How MEMCOM works

To gain a better understanding of the MEMCOM - C-API it is best to learn how the MEMCOM system operates. A MEMCOM database file consist of contiguous blocks of data called datasets. Once a dataset is created its size cannot be changed. All new datasets are appended at the end of the file. Each dataset can be equipped with a descriptor. A descriptor is a separately stored dataset (relational table) that holds additional informa-

tion about the dataset on the database to which it corresponds. When a database is opened, the datasets are not yet loaded into memory. Even when the properties of a dataset are inquired after, the set itself is not loaded into memory. Only when a client program calls for retrieval of the data, MEMCOM reads the data from the disk and places it in a reserved piece of memory or it reserves the memory itself and returns the pointer to the newly allocated memory block containing the data. This operation can be done relatively fast because all data on the disk is contiguous (and not scattered like in some database systems). Because the size of the datasets can not be changed there is no possibility to really remove a dataset from a database. Instead, the set is flagged deleted, but the disk space used for it can not be reclaimed. (A smaller dataset that carries the same name can overwrite the old dataset. The old set size remains intact, however, and the remaining space is filled with zeros.) To offer a way to tidy up databases which are heavily littered with deleted datasets, a defragmentation function is available. (It is however more advisable to copy the datasets of interest into a new database.)

### 5.1.2 Data structures and supported data types

A database consists of datasets and descriptors, as mentioned above. Each dataset has a fixed number of attributes which are defined in the header of the database file. The attributes are: set name, number of dimensions, size of each dimension, total size of the set, address of the data and the size of the accompanying descriptor. A similar group of attributes exist for the database file itself. The attributes can be obtained in the form of a variable of type `struct`. The actual data is stored in one contiguous block on another part of the disk.

The data stored in the ordinary datasets can be of the following data types:

- I Integer format (32 bit)
- J Long integer format (64 bit)
- E Real floating-point format (32 bit)
- F Floating-point format (64 bit)
- D Double-precision floating-point format (64 or 128 bit)
- K Character (8 bit)

- $ Relational table
- ! Array table
- U Unspecified format (same as void) (32 bit)

As mentioned before, it is not practical to try to support all different data types in the MEMCOM - MATLAB coupling, since MATLAB has a very poor support for most of these data types. First the coupling will be designed to work with the types `int`, `double` and relational tables[1]. Special attention most be paid to the last data type mentioned. The data of this type (unlike the other types) is stored in position-independent format, i.e. elements of the tables may be stored, retrieved, added or deleted in any order with the use of keywords. A relational table can contain data of most of the other types, such as `int`, `double` or `char`. The descriptor table available for each dataset is actually a relational table. Thus, it is quite sensible that a number of functions that operate on relational tables are also used to perform the same operation on descriptor tables.

## 5.2 Features of MATLAB and mex-functions

In this section is explained how mex-functions work and how they should be used. Thereafter, the data structures and data types of MATLAB are explored. The first subsection, however, comments on some important MATLAB features. (See [3])

### 5.2.1 important MATLAB features

**mxArray** All data created or processed in MATLAB is contained in so-called mxArrays. These arrays can be of a number of different types which will be described in section 5.2.3. Each type has its own set of properties and own data structure. The best supported type of mxArray in MATLAB is the (two dimensional) numerical matrix of type `double`. Almost all functions support this type mxArray, where the use of the other types is in most cases somewhat limited.

[1] The attempt is made to make the addition of other data types in a later stage of development easy. This is done by clustering type specific data in IF- or SWITCH-blocks, which can easily be extended with code to support other data types.

**workspace** MATLAB keeps track of the mxArrays generated in scripts or from the command line. All mxArrays in use are stored in the MATLAB workspace. In principal, MATLAB functions can only operate on mxArrays that are present in the workspace or parsed to them in the function call.

### 5.2.2 How mex-functions work

When a mex-function is called from the command line or from a MATLAB script, the mex-function gains control of the MATLAB application. A pointer to an array of input mxArrays and a pointer to an array of output mxArrays is parsed to the mex-function. The standard input and output are connected with the MATLAB prompt. Two special sets of functions (mx-functions and mex-functions) are available to manipulate the input mxArrays or mxArrays created by the function. It is also possible to create, delete or manipulate mxArrays in the MATLAB workspace that were not parsed to the functions, but one can easily understand that this is confusing in most cases. If the mex-function has output arguments, then the mxArrays containing the output data can be assigned to them by using the pointer to the output array. (Notice that an mxArray created in a mex-function will not appear in the workspace. Only mxArrays which were assigned to the output array will appear in the workspace, but not before the mex-function is terminated and the control returns to the MATLAB prompt.)

### 5.2.3 Data structures and supported data types

MATLAB has an elaborate scheme of data structures which is manly focused on mathematical applications. All data in MATLAB is contained in array's. These so called mxArrays can be of a variety of types. A short summary of the available data types is given below. Note that most types can be multi dimensional.

- cell array - a structure in which each element may be any of the other array types (including cell arrays).
- character array - a structure containing character strings as elements
- struct array - an array similar to the `struct` type in the C language. It can contain elements of any type except of type cell. This data type is always one dimensional.
- numerical array - a structure in which each element is of the same numerical type.

The following numerical types (or classes) are available: `sparse`, `double`, `single`, `int` (8 bit), `unsigned int` (8 bit), `int` (16 bit), `unsigned int` (16 bit), `int` (32 bit) and `unsigned int` (32 bit). It can be seen that it is difficult to make a strict separation between data structures and data types because these two concepts are more or less interwoven in MATLAB. If, for instance, one chooses to use the data structure of a sparse matrix, there is no longer a choice in the data type, because it is `double` by default.

A big advantage of the MATLAB data structures is that it enables the writer of a mex-function to make use of the MATLAB memory management system. Allocating new memory (if necessary) with the special MATLAB functions `mxAlloc` and `mxMalloc`, and reassigning of pointers with the `mxSet` functions, assures that MATLAB will free memory once it it is no longer used. It is also possible to make memory persistent, so MATLAB will not attempt to free it when it is no longer used within MATLAB, but this makes the calling mex-function responsible for freeing the memory.

### 5.2.4 name conventions

The names of the mex-functions of the MEMCOM - MATLAB interface are derived from the names of certain C-API functions that perform similar tasks. In case the mex-function is overloaded, the C-API name that describes the task of the function best is chosen. The main reason for approaching the function names in this way, is to keep a certain sense of unison. The set of mex-functions are in a way a new MEMCOM - API, so it makes sense to adhere to the name conventions already established. Users which are already familiar with the other APIs, will be able to work with the new functions straight away.

Almost all functions in the C-API begin with the letters mcDB or mcRT (which stands for either MEMCOM database or MEMCOM relational table). This is also the case for the mex-functions. The function names of the C-API were chosen such, that they describe the task carried out by the corresponding function in a few words. Capital letters are used to emphasize the separate words. This approach was adopted when naming the mex-functions. Notice that MATLAB is case sensitive, so all letters in scripts or commands most be in the right case.

## 5.3 Features of B2000

This section describes in short some of the data structures used in B2000 and the impact they have on the MEMCOM - MATLAB coupling. Thereafter, the name conventions that apply to the dataset names used in B2000 are commented on. (See [4] and [6])

### 5.3.1 Data structures

The B2000 system makes extensive use of the MEMCOM Fortran-API and C-API. The data structures used in B2000 resemble to a great extend the data structures of the MEMCOM system. There are, however, a number of data structures in B2000 databases that are not inherent to the use of the MEMCOM system.

To reduce the storage capacity needed to hold the sometimes large system matrices of a finite element calculation, two methods are frequently applied. The method most used to store a large symmetrical matrix is the skyline method. For non-symmetrical matrices, compressed row storage or the sparse method is mostly used. Skyline matrices or sparse matrices are represented on a MEMCOM database by respectively two or three datasets. MEMCOM offers no facilities to connect or relate more datasets with each other. The sets are generally associated with one another through name conventions.

MATLAB can work with either sparse or full matrices. It is not advisable to construct a full or a sparse matrix out of more datasets in the MATLAB environment. MATLAB is not designed for such a task. (For instance the extensive use of for loops slows MATLAB down considerably.) It is evident that a coupling between MATLAB and a MEMCOM database created with B2000 must offer a way to read the datasets of a skyline or sparse matrix on a MEMCOM database and construct the corresponding matrix in MATLAB. The coupling should also make it possible to write matrices created in MATLAB to a MEMCOM database in skyline or sparse format.

**Complex numbers** There are a number of ways to store (skyline or sparse) matrices with complex numbers. Until now, no standard was chosen for the B2000 system. There are two ways to approach this issue: If there are far less imaginary non-zero elements then real non-zero elements in the matrix, the following approach is the most efficient one when it comes to memory usage and storage space. The real and imaginary part are completely separated and the two resulting matrices are stored

completely independent from one another. A disadvantage of this method is that it results in slow access time to the different elements. The reconstruction of a full matrix has to be performed not once but twice. The second method considers the complex number as a whole and only the values of the two parts are stored on two separate datasets or interlaced on one dataset. The second method, with the element's real and imaginary part stored interlaced, is most used in other applications (such as solvers), since both parts are generally needed at the same time. The choice is made to adhere to this standard.

### 5.3.2 Naming conventions

As mentioned above, the datasets that make up a skyline or sparse matrix can only be associated with each other through naming conventions. The principal idea behind the naming conventions of the skyline and sparse matrices is the same. The set name consists of a base part which identifies the matrix and a second part that identifies the function of the set. The layout and the conventions of both matrix types are explained in the next paragraphs.

**cycles** In a number of B2000 applications, the dataset names on the MEMCOM database are subjected to a method to organize the data of cyclic calculations. The number of the cycle when the database was created or used is generally appended to the end of the set name. Even in most cases where there is no need for the use of cycles, the cycle number zero is appended to the set name. The MEMCOM - MATLAB coupling would benefit from the possibility to make use of cycles when transferring data from and to a MEMCOM database.

**skyline matrices** The principal idea behind the skyline notation is that only the data which lies in the skyline of the matrix is stored. Since the notation is only applicable to symmetric matrices, only the upper triangle of the matrix is regarded.

To find the skyline of this triangle, one strips all contiguous zero-valued elements starting from the 'top' of the matrix. The remaining data elements of each column are stored in one contiguous dataset. This dataset is called the 'global' dataset and the name convention used is: `setname.GLOB.x`. The 'x' is an integer value that indicates in which cycle the set was created or used.

A second dataset contains for each column the address on the 'global' dataset where the data of that column starts. This dataset is thus revered to as the address dataset and the following name convention is used: `setname.ADR.x`. Again the 'x' points to the cycle of the dataset.

**sparse matrices** No naming conventions for this matrix type has been developed yet for use in B2000. The choice was made to adhere to the conventions most used in other applications. The concept of this notation is to store only the non-zero elements of a matrix.

All non zero data elements are stored in one contiguous dataset. Since this dataset holds the values of the elements it is called the 'value' dataset. The name convention used is: `setname.VALUE.x`.

A second dataset contains for each row the address on the 'value' dataset were the data of that row starts. Since that dataset is an index of pointers, it is often revered to as the 'pointer index'. The name convention used for this dataset is: `setname.PTRINDEX.x`.

The third dataset also consists of pointers. For each row, the values of this set points to the column where a data element is located in that row. The dataset is revered to as the 'row index' hence the name convention: `setname.ROWINDEX.x`.

**Example** A system stiffness matrix SVAR that is stored in sparse form and was created or used in cycle 2 consists out of three sets with the following set names:
`SVAR.PTRINDEX.2`, `SVAR.ROWINDEX.2` and `SVAR.VALUE.2`

# 6

# Defining the mex-functions

This chapter deals with a number of general design decisions and two important concepts, namely function libraries and error handling. In the last section is describes how the the different task the coupling is required to perform are grouped together, to form functions which the user can call from within MATLAB. Lastly, a number of general remarks that apply to the implementation of the MEMCOM - MATLAB coupling are stated.

## 6.1 General design decisions

The mex-functions that are to be designed are in a sense a new MEMCOM - API. The question rises to what extend the new API should resemble the already existing APIs. It is clear that a number of API-functions can be almost directly converted to functions of the MATLAB - API. But the concept of the languages for which an API already exist and the concept of MATLAB is somewhat different. MATLAB is in first instance a language in which programming is done from the command line. This gives rise to the need for short and efficient programming capabilities. One way to reduce the amount of code is to group functions that are (almost) always carried out after each other. For instance, verifying the existence of a dataset and loading its content, is carried out by a single function. Grouping functions in this way is also a method to shield of the user of the coupling from MEMCOM error messages. Another way to endorse short and efficient programming and to make code more readable is to make use of function overloading.

### 6.1.1 Writing a function library

The internal MATLAB C-compiler works like any other C-compiler. This means that linking other C-libraries is possible. The ultimate reason to develop a function library is of course code re-usability. Since a number of similar tasks have to be performed in most of the mex-function, it makes sense to develop a function library. It is hard to comment in advance on the content of the library. As the mex-functions are implemented, the need to move pieces of code to the library so they can be used by more mex-functions will arise. The library is reviewed in the last section of appendix A.

### 6.1.2 Error handling

As mentioned above, MATLAB is a language that is mainly programmed from the command line. This does not only require short and efficient programming capabilities, but also a proper error handling. Since MATLAB is a scripting language, all errors that occur are runtime errors. Most MATLAB programmers depend heavily on the error massages MATLAB returns. It is therefore important to pay special attention to the error handling of the mex-functions used for the coupling.

A property of MATLAB's error handling is that it is rare that a function returns its status. In almost all cases when a script is run, a function either succeeds, and the execution of a script continues or the function fails, an error message is displayed and the execution stops. This approach was followed when designing the error handling of the mex-functions. In short, when a function cannot perform its task for a certain reason, the function issues an error message and the command returns to the MATLAB prompt.

An attempt is made to present the user of the coupling with very specific error messages. This approach differs from the way the MEMCOM error handling was set up, so extra effort is made to shield of the user from MEMCOM errors.

## 6.2 Implementation of the mex-functions

### 6.2.1 Assessment of the required mex-functions

The mex-functions that are needed to work with MEMCOM databases from within MATLAB can be divided into three groups. All available functions and groups and a short explanation of their working are listed below.

**The first group** consists of two functions that are responsible for opening and closing a MEMCOM session and the file handling.

- `mcDBopenFile` Start up a MEMCOM session and open a database.
- `mcDBcloseFile` Close one or all databases or shut down MEMCOM.

**The second group** of functions is concerned with the transfer of data to and from MEMCOM.

- `mcDBgetSet` Load a MEMCOM dataset into MATLAB.
- `mcDBgetDesc` Load a descriptor of a dataset into MATLAB.
- `mcDBputSet` Write a MATLAB variable to a MEMCOM dataset.
- `mcDBputDesc` Write a MATLAB struct to a MEMCOM descriptor.

**A third group** of functions is made up out of functions that are involved in database management.

- `mcDBinqSetAtt` Returns a `struct` containing attributes of a set.
- `mcDBdeleteSet` Remove a set from a MEMCOM database.
- `mcDBdeleteDesc` Remove a descriptor from a MEMCOM database.
- `mcDBpack` Defragment a MEMCOM database.

The structure of the different subtasks of a function that have to be carried out in order to accomplish a certain main task, is reflected in the way the function structure was conceived. It may therefore be insightful to learn not only about the working of the available mex-functions, but also about their layout and the auxiliary functions they use to perform their tasks. The working and implementation of the functions listed above and the function library that was developed are covered in appendix A.

### 6.2.2 General structure of a mex-function

This subsection describes the general layout of the implemented mex-functions. The basic layout is practically the same for all functions. The main function body that is run from a the file containing the mex-function must begin with the following line:

```
mexFunction(int nOutputarg, mxArray *Outputarg[],
            int nInputarg, const mxArray *Inputarg[])
```

The first operation the main function performs consists of verifying the number and type of input and output data. To check the number of input and output arguments, the variables `nOutputarg` and `nInputarg` parsed to the function are used.

Thereafter, the type of the input arguments is checked using several mx-functions. These functions are designed to operate on or probe mxArrays. Note that the output types can not be checked because there are no output arguments parsed to the function. Instead a pointer to an empty array that is large enough to hold pointers to the output mxArrays is parsed. If all went well, the values of the input arguments must be extracted from the mxArrays since C does not support these data types. Most routines that extract data from mxArrays are placed in the function library, because they are used by almost all functions.

The specific layout of a certain function depends on the task the function has to perform. If a minor task has to be carried out, the implementation is usually contained within the main function body. If the function is overloaded or has to carry out an extensive task, the choice is made to place the implementation of these task in subroutines and call them from the main function body. When a function has output arguments, they are usually created and assigned to the output array as the last step in the main function.

# 7

# Evaluation and recommendations

## 7.1 Evaluation

In this section the MEMCOM - MATLAB coupling that was developed is evaluated. First, it is discussed to what extend the requirements stated in subsection 3.2 are met. Thereafter, the requests are subjected to the same scrutiny.

### Requirements

For the sake of clarity, the requirements of the MEMCOM - MATLAB coupling are stated once more below.

- Read and write vectors or full two dimensional matrices, stored on a MEMCOM database, from and too MATLAB while maintaining the size and dimension.
- Read and write descriptor tables, stored on a MEMCOM database, from and too MATLAB
- Read and write relational tables, stored on a MEMCOM database, from and too MATLAB
- Vectors and Matrices must be of type `double` (64 bit) or `int` (32 bit)
- Information in descriptor tables must be of type `double`, `int` or `char`

- Read and write two dimensional matrices in skyline or compressed row storage format, stored on a MEMCOM database, from and too MATLAB

All requirements are met without any compromises. The coupling is suited to facilitate a basic coupling between MEMCOM and MATLAB and what is more, it makes the transfer of matrices stored in skyline or sparse form possible between the two programs.

## Requests

The requests of the coupling are listed once more below:

- Read and write more dimensional matrices (up to five, the limit of MEMCOM) from and to MATLAB.
- Read and write vectors and matrices of other types than `double` or `int` (MEMCOM supports among others `real`, `char` and `unknown`)
- Expand the number of types that are allowed in descriptor tables and relational tables to the same number of types that MEMCOM allows.

While working on the MEMCOM - MATLAB coupling it became more and more evident that most of the requests made, where either not feasible without generating an extensive amount of extra code, or would not add much to the usability of the coupling.

The expansion of the number of supported types with the data type `real` (32 bit), means duplicating existing code and making minor adjustments for this new data type. Since this data type is very poorly supported in MATLAB this addition would not be of much use. Long integers (64 bit) and 128 bit double-precision floating points are not supported at all in MATLAB and can thus not be supported in the MEMCOM - MATLAB coupling. The unspecified format can not be supported, because the data type is not known in advance. The routines that convert the `unknown` data types to mxArrays and back can thus not be written. If one wishes to use such a data type, it is best to write a specific mex-function that handles that type. The Array tables can be added relatively straightforward, because a MATLAB equivalent is available: the cell class mxArrays. Since array tables are rarely used at present, the efforts to implement the routines that handle this data type are not justified. The most natural expansion of the supported data types would be the support of character strings. The data type is already supported in relational tables and it could be implement without much effort to work also with ordinary datasets.

The choice was made not to expand the number dimensions that is supported to more than two. Though MATLAB supports matrices with more than two dimensions, the feature is mainly used for storage. If one wants to operate on such matrices, it is necessary to split them up into more matrices with less dimensions. On top of that, the feature is rarely used in MEMCOM.

## 7.2 Recommendations

The recommendations which can be made on the basis of the evaluation above and recommendations on how to put this coupling to use are presented in this section.

### improvements to the new MEMCOM - API

- Expand the support of available data types with the character string data type.
- Implement functions to probe certain properties of database files. These functions should make it possible to check for instance the type and existence of files or the number of sets on a database without risking error messages and abortion of a running MATLAB script.
- Design a GUI with the help of GUIDE, MATLAB's graphical user interface development environment. The GUI should allow for examination of databases on the disk and data transfer between MEMCOM and MATLAB. It should resemble the existing MEMCOM browser.
- Offer the option to store and retrieve the arrays of which skyline and sparse matrices consist as zero based or non zero based arrays. (see subsection 9.1.1)

### Applications of the new MEMCOM - API

- Rapid implementation and testing of new algorithms.
- Make use of MATLAB math functions and matrix manipulation techniques to avoid implementing complex algorithms in the first stages of the design of new B2000 processors.

- Use the coupling for educational means to demonstrate the working of B2000. Students can work with B2000 and all the processors, inspect data, generate simple pieces of extra code that can manipulate the input- and output data and will thus experience the workings of B2000 first hand. A main advantage is that the students will only need to know how to program in MATLAB. Experience in other programming languages is not necessary.

# Part II

# An iterative eigenvalue solver in MATLAB

# 8

# Viscous elements in B2000

The second task of the assignment is to integrate the elements 95 and 96 once more in B2000, solve the eigenvalue problem in MATLAB and compare the results with the results obtained with the old B2000 solver.

In the first section of this chapter a brief introduction to the viscous elements 95 and 96 and their corresponding coupling elements 104 and 105 is given. Thereafter, the task description of the second subtask and the specific needs and requests to the iterative solver are explored.

## 8.1 Introduction to viscous elements 95 and 96

Elements 95 and 96 are so called viscous elements. They account for the viscous and thermal effects on the propagation of pressure waves. Viscous elements have pressure degrees of freedom unlike structural elements which have displacement degrees of freedom. Coupling elements (104 and 105) are necessary to be able to use viscous elements in combination with structural elements.

In chapter five of the thesis of W.M. Beltman [1] is described how the system matrices of a model that involves structural, viscous and coupling elements are build up and how an eigenfrequency calculation might be carried out. It is made clear that an eigenvalue solver for complex, asymmetric and frequency dependent matrices is needed to be able to obtain the eigenvalues.

## 8.2 Task description

### 8.2.1 Subtasks

The B2000 package has modular nature. The different steps of a finite element calculation are captured in separate processors or control modules. It is possible to run an single module or to run multiple modules in one session from a macro processor. The code that constructs the element matrices of elements 95 and 96 and the corresponding coupling matrices 104 and 105 was already available and contained few bugs.

The code that once assembled the system matrices of these asymmetric and complex element matrices is no longer available in B2000. Thus, an assembler for sparse complex system matrices has to be developed. The design and implementation of this assembler is described in chapter 9.

The eigenvalue value problem that the iterative MATLAB solver most solve, is not a standard one. The matrices are not only asymmetric and complex, but the mass matrix of the system is dependent on the eigenvalues that are to be solved. An older version of B2000 was equipped with a solver for eigenvalue problems concerning asymmetric complex matrices, but a guess of the eigenvalue that was to be solved had to be presented to the solver as input. To make the eigenvalues converge to the actual eigenvalues, the solver had to be restarted many times by hand, each time presenting the solver with the previously calculated eigenvalue. A disadvantage to this approach is that the first guess of the eigenvalue must be relatively accurate to insure convergence to the desired eigenvalue.

The code of this old solver is no longer present in the current version of B2000. This was the main reason to attempt to solve the eigenvalue problem in MATLAB, since MATLAB is equipped with a similar eigenvalue solver. The newly developed MEMCOM - API for MATLAB makes it possible to access and manipulate the database on which the system matrices and other variables of the problem are stored. Another great advantage of using the MATLAB environment, is that a script or function can perform the iteration, once carried out by hand, fully automatic.

### 8.2.2 Requirements and requests

The requirements and request to which the integration of the elements and the iterative solver are subjected are listed below.

**Requirements**

- Use code already available in B2000 - 2.5 when possible.
- Adhere to the standards when it comes to B2000 conventions.

The iterative solver in MATLAB gives rise to an extra requirements:

- The solver most return the converged eigenvalues, the corresponding eigenmodes and the relative error.

**Requests**

- The iterative MATLAB solver should operate as a function and not as a static script.
- Settings of the solver, such as tolerance and maximum number of iterations, are not hard coded in the script, but can be parsed to the solver in a function call.

# 9

# Assembler for asymmetric complex matrices

The design and implementation of a system matrix assembler for asymmetric complex matrices is described in this chapter. First, the requirements of such an assembler are dealt with. Thereafter the layout and implementation of the assembler is commented on. In the last section the design and working of an utility program, that was devised to compare the outputted data of the assembler with the system matrices outputted by B2LIN, is described.

### Requirements

The requirements of the asymmetric complex matrices assembler are listed below.

- The assembler most handle asymmetric complex element matrices.
- The assembler most handle all storage types in which B2000 stores its element matrices (diagonal, full or upper triangle).
- The output of the assembler most adhere to standards of available solvers.

## 9.1 Design decisions and boundary conditions

### 9.1.1 Data structures

Before any other design step can be taken, the data structures of the B2000 element matrices and the input format of available solvers have

to be studied.

The element matrices out of which the system matrix is assembled are stored in three ways in B2000. The first method is to store all data elements of a matrix row-wise. This method is generally applied to asymmetric matrices. The second method is used for symmetric matrices. Only the data elements in the upper triangle (including the diagonal) of the matrix are stored. The last method is applicable to lumped matrices, so only the data element on the diagonal of the matrix are stored. B2000 stores the real and imaginary parts of complex numbers in a separate datasets which often have a similar name (see subsection 9.1.3).

The input data that is presented to a solver for complex non-symmetric eigenvalue problems, must be in sparse format. The are no standards yet for this format in B2000[1]. The choice was made to adhere to the standards most used in other mathematical applications. A consequence of this decision is that the different arrays used to store a matrix are zero based. This is a deviation of the B2000 standards for skyline matrices. The decision also implicates that the MEMCOM - MATLAB coupling treats the arrays of a skyline matrices as non-zero based and the arrays of sparse matrices as zero based. (It is more elegant if the user is offered the choice to store and retrieve a matrix in zero-based-mode or non-zero-based-mode)

### 9.1.2 Complex numbers

The implementation of some functions in the MATLAB coupling would be simplified if the real and imaginary part of complex numbers were stored in separate datasets, since MATLAB internally stores its data this way. However, most applications that handle complex numbers store the real and imaginary parts interchanged in one contiguous data string. To avoid the assembler to be restricted to work only in combination with MATLAB, the choice was made to store the real and imaginary parts of complex data interchanged and not on separate datasets.

### 9.1.3 Name conventions

The name conventions used for the names of the system matrices were already explained in subsection 5.3.2. The names of the element matrices are subjected to the following name conventions (See [4]).

[1] The naming conventions of this format in B2000 are described in subsection 5.2.4

**Base name**

- The first part of the set name is either ELSV, EMDV, EMSS.
- If the set contains the imaginary data of an element matrix, the last letter of the names above is replaced by an I (ELSI, EMDI, EMSI).

**ELSV or ELSI** indicates an element stiffness matrix. It contains depending on the size, the elements of the full matrix or the upper triangle of the matrix.

**EMDV or EMDI** indicates an element mass matrix. It contains only the diagonal elements of the matrix.

**EMSS or EMSI** indicates an element mass matrix. It contains all elements of the element matrix.

**indexes** The base name is followed by four indexes separated by points.

- The first number indicates the branch to which the elements in the set belong.
- The second number is unused and zero as default (B2000 heritage).
- The third number indicates the internal element type number.
- The last number indicates the cycle in which the set is created or used.

**Example** The set name `ELSV.3.0.95.2` indicates a set that holds the element stiffness matrices of all elements with type 95, that can be found in branch 3 and were created or used during cycle 2.

### 9.1.4 Choosing a language

The languages that are supported as a standard in the B2000 makefile are Fortran, C and C++. The choice was made to program the assembler in C++. The main reason for this is the availability of objects in C++. An object called `b2aemsacDataset` is created which can hold all data for a matrix of type skyline or sparse. The object feature is not used to its full

advantage, but it helps to form a clean data structure and improves code readability. The constructor is used to make sure that a number of member variables are properly initialized, and the destructor of the object insures that memory allocated to store the large amounts of data is freed, when the object is no longer needed.

The same tasks could have been performed in either Fortran or C, but the code would be less readable. The program would also be harder to maintain and appending new features or extensions to it would be much more error prone.

## 9.2 Implementation

All processors in B2000 consist of a control module and a main function. The main function makes it possible to call the processor from the command line and causes it to work as a stand alone program. The control module is the code that does the actual work and is often stored in a separate file. This way, the control module can be called from the command line or from within other programs or macro processors.

The same concept was applied to the asymmetric complex system matrix assembler called `B2aemsac` (B2000 Assembler for Element Matrices to Sparse Asymmetric Complex form). In the next subsections the implementation of the different functions is commented on. The attempt was made to outline the task and structure of the created functions, without concerning with the details of the implementation.

### 9.2.1 The main function

All main functions in B2000 carry out the same tasks:

- Handle all input arguments parsed to the program at start up or read PCL[2] commands.
- Start a MEMCOM session and open the requested file.
- Call the appropriate control module(s) with the appropriate arguments.
- Close all files and close down MEMCOM.

[2]PCL is the abbreviation for the Processor Command Language. The language is used when a processor is called without arguments.

### 9.2.2 The control module

The code of the control module is relatively compact, since a big part of the code resides in the member functions of the matrix object. First, an instance of the matrix object is created. The function retrieves the number of the branches that exist on the database and iterates through them. For each branch, the function iterates through the element types used and for each combination of element type and branch the elements present are added to the system matrix. After all elements are added, the vectors containing the row indexes are sorted and the sparse system matrix that exist in memory is written to the MEMCOM database. All allocated memory is freed by the destructor of the matrix object when the function returns to the calling program.

**The `colData` structure and `vector` class** The data structure `colData` of type `struct`, is used to easily manipulate the data elements of the matrix. The `struct` consists of three variables that contain the real and imaginary part of the value of a data element and the number of the column where it resides. For each row in the system matrix a vector, that holds elements of this data type, is created. As more elements are added to the system matrix, the vector that holds the elements most increase in size. The sparse method requires that the data elements are stored such, that within each row the column number of the data elements increases. This means that the row vectors containing the data elements, have to be sorted according to column number after all element matrices are added. The `vector` class of the Standard Template Library offers a vector type with this functionality.

**The matrix object `b2aemsacDataset`** All functions operating on the member variables of the object are implemented in-line. Since there are no plans to use the devised object outside this program, there is no need to protect the implementation of these functions by means of encapsulation. All member variables are kept private. Their values can only be obtained or changed by calling a member function that operates on them. This approach was taken to prevent (un)accidental alterations of the values of the member variables.

The following list of functions are public and can be called from anywhere in the program to operate on the data of an instance of the matrix object:

- `getBranches` loads the dataset that holds the numbers of the existing branches and returns.
- `getBranchNumber` returns the number of a requested branch.
- `getNumberOfBranches` returns the (active) number of branches in the model.
- `getElementTypes` loads the table with the numbers of all used element types.
- `getNumberOfElementTypes` returns the number of different element types used in the model.
- `getNumberOfElements` returns the number of elements used of a specific type in a specific branch.
- `addElement2Matrix` calls one of three private member functions that add an element matrix to the system matrix.
- `sortVectors` rearranges the order in which the data elements of the vectors, that contain the row and value information, are stored
- `write2MemCom` writes the different datasets that make up the sparse matrix to MEMCOM.

The matrix object also harbors a number of private functions. These are involved in memory management, retrieval of datasets and the actual addition of data to the system matrix. It is not appropriate to dwell on the details of their working and implementation.

## 9.3 B2COMPARE

### 9.3.1 B2COMPARE as an utility function

To compare the output of the sparse complex matrices assembler with the output of B2LIN, an utility program was written. B2LIN produces matrices in skyline format. B2AEMSAC produces matrices in sparse format. Thus, a program that is to perform the comparison between the outputs, should be able to extract random data elements from a skyline or a sparse matrix.

At first the attempt was be made to generalize the working of the program as much as possible. In that way the program can be of service to

other B2000 developers in the future. To make the program less type specific, it should not only be capable of comparing a skyline matrix on one database with a sparse matrix on another database, but also for instance, a skyline matrix with another skyline matrix or a sparse matrix with a full matrix that are stored on the same database. It should also be possible to compare parts of matrices instead of entire matrices.

All options must be parsed to the program from the command line, however, and it becomes a huge task to handle all input arguments, or write a textual interface. Since the program is but an utility program and was not mentioned in the task description of the assignment, it was ultimately decided to drop the objective of generality.

### 9.3.2 The working of B2COMPARE

The working of the program is as followed: The program is called with one or two filenames of MEMCOM databases. It opens the database(s) or asks the user to input a filename when none was specified. If two filenames are parsed, the program tries to detect the system matrices independent of the storage format (full, skyline or sparse). It will only try to compare the mass or stiffness matrices if they are present on both databases. If the program is called with one filename, it will try to detect mass and stiffness matrices in different storage formats. When more than one format is present for a matrix, the program will perform a comparison. If a comparison could be made, the program presents the following output data for each matrix: The storage type and size of the matrix and the number of inequalities encountered in the data elements.

# 10

# Results, evaluation and recommendations

## 10.1 Results

### 10.1.1 Element and system matrices

B2COMPARE (see section 9.3) was used to validate the output of B2AEMSAC by comparing the results of the new assembler with the results of B2LIN. This validation was insufficient since only the construction of symmetric none complex system matrices could be tested. The old asymmetric complex B2000 solver stores the imaginary part of the data in separate matrices (Notice that the signs of all values are inverted). So, the system matrices produced with B2AEMSAC were compared by hand[1] with the matrices produced by the old assembler/solver.

**Relative errors** The biggest relative error found was $1.5\%$. All elements of the imaginary part of the system mass matrix deviated from the corresponding elements of the system mass matrix calculated by the old solver with this factor. The real parts of the elements in the mass matrix deviated with a factor of $0.006\%$. These errors are, however, not due to the assembling process. The calculated element matrices of both assemblers where also compared and the same factors were found for the imaginary and real part of all element mass matrices.

For all other calculated values (both in the element stiffness matrices and the system stiffness matrix) the relative error was smaller than $10^{-14}$. Relative errors of this magnitude are commonly accepted since they can

[1] The comparison of the different matrices was carried out with the help of MATLAB.

result from a change in the order of calculation steps in combination with rounding errors due to a finite machine precision. Relative errors in the order of $1\%$ are, however, not accepted.

**Element code** After further investigation there seemed to be a discrepancy between the equations used in the code for calculating the mass matrix of element 95 and equations in the thesis of Marco Beltman. A number of adjustments were made to the original equations to be able to use the same coupling elements for the mass and stiffness matrix and by a difference in definition of the parameter $h_0$[2].

The deviation in the values of the calculated element matrices is, however, not explained by the adjustments that were made to the code. Since the original code that was used to compile the binary version of the old solver is not available, it remains unclear what the reason for these differences is.

The impact of deviations in the imaginary part of the mass matrix on the calculated (converged) eigenfrequencies was examined. When all values in the imaginary part of the mass matrix deviate as much as 50% (with respect to a previously calculated mass matrix), the converged eigenfrequencies only deviate $\pm 0.070\%$.

### 10.1.2 Converged eigenvalues

The results produced by the iterative MATLAB solver must be validated by comparing them with the results made by manually iterating the eigenfrequencies with the old solver. The first three converged eigenfrequencies where calculated with both methods and are listed below in table 10.1.

| *Frequency* | *Old solver* | MATLAB *solver* | *relative error* |
|---|---|---|---|
| | $A$ | $B$ | $\|A-B\|/\|A\|$ |
| $\omega_1$ | $79,0864 + 2,45016i$ | $79,0873 + 2,45101i$ | $1,56 \cdot 10^{-5}$ |
| $\omega_2$ | $152,813 + 1,81586i$ | $152,813 + 1,81630i$ | $2,88 \cdot 10^{-6}$ |
| $\omega_3$ | $226,263 + 2,35636i$ | $226,264 + 2,35685i$ | $4,92 \cdot 10^{-6}$ |

Table 10.1: Eigenvalue comparison

[2] In the Thesis of Beltman, $h_0$ represents the half of the thickness of the viscous layer, while in B2000 $h_0$ represents the entire thickness of the viscous layer

## 10.2 Evaluation

In this section the sparse system matrix assembler and the iterative MATLAB eigenvalue solver that were developed are evaluated. First, there is discussed to what extend the requirements stated in subsection 8.2.2 are met. Thereafter, the requests are subjected to the same scrutiny. In the last subsection the results produced with the new iterative MATLAB eigenvalue solver are evaluated.

### 10.2.1 Requirements and requests

**Requirements**

- Use code already available in B2000 - 2.5 when possible.
- Adhere to the standards when it comes to B2000 conventions.

The iterative solver in MATLAB gives rise to an extra requirement:

- The solver most return the converged eigenvalues, the corresponding eigenmodes and the relative error.

The first requirement was met to a great extend. The old element code could be adopted with relative ease and the element matrix assemblers, B2EP and B2MP, could be used unaltered. A system matrix assembler, however, was not available in B2000 and had to be developed.

The B2000 standards and naming conventions were respected in the implementation of the new code. The where no standards and naming conventions available for sparse matrices in B2000, the standards used in most other solvers and applications were adhered to.

The iterative solver returns the converged eigenfrequencies, the corresponding eigenvectors (or eigenmodes) and the relative error based on the last two iteration steps.

**Requests**

- The iterative MATLAB solver should operate as a function and not as a static script.
- Settings of the solver, such as tolerance and maximum number of iterations, are not hard coded in the script, but can be parsed to the solver in a function call.

The iterative solver was implemented as a MATLAB m-function. The first estimates of the eigenfrequencies, the thickness of the viscous layer and settings of the solver can be parsed to the function as input arguments. A number of default settings, such as tolerance, maximum number of iterations and frequency shift, are hard coded in the function. All settings can be altered by parsing an option structure to the function.

### 10.2.2 Evaluation of the results

Despite the relative error of $1.5\%$ in the imaginairy part of the system mass matrix, the results of the itterative MATLAB solver correspond with the results of the old solver to great extend. The methods and algorithmes used clearly suffice, but verification of the element code used is required.

Notice that when using the old solver, the the calculation of the fourth eigenvalue is not straightforward. The first estimation has to be very precise in order for the solution to converge. The iterative eigenfrequency solver in MATLAB does not suffer from this problem. All eigenfrequencies are calculated with the same ease and the first estimation of the fourth eigenfrequency does not have to be more accurate than that of the other frequencies.

## 10.3 Recommendations

The discrepancy between element and system mass matrices calculated with the new assembler and calculated by the old assembler/solver, give rise to the need to verify the element code currently used.

A second recommendation rises from the wish to enable Fortran users to use the sparse system matrix assembler in a more convenient way. The arrays that store the pointer index and row index are zero based at present. Since Fortran uses non-zero based arrays, it would be good practice to offer the option to store and retrieve the arrays of which sparse matrices consist as zero based or non- zero based arrays (see subsection 9.1.1).

# 11

# An iterative solver in MATLAB

The requirements and request to an iterative solver were already listed in subsection 8.2.2. The next section deals with the general layout of the solver. Thereafter, the working of the solver is commented on.

## 11.1 Layout of the solver

### 11.1.1 Function blocks

The solver is programmed as a MATLAB function that is overloaded and takes up to four input arguments and returns up to three output arguments. The function consists of two main blocks.

The first block of the functions is used to handle the input arguments and declare the size and type of the output arguments. In the second block, the function attempts to converge to frequencies that are in the vicinity of the estimated eigenfrequencies. When the iteration process is stopped, the function provides the user with feedback of the results. The iterative process is described in the next subsection. The handling and nature of the input and output arguments is commented on in section 11.2

### 11.1.2 Iterative process

The following iterative procedure is followed for each estimated eigenfrequencies. The database is opened and the estimated frequency (and thickness of the viscous layer) is inserted into it. Then the B2000 processors B2EP and B2MP are called to calculate the new element matrices. B2AEMSAC is called next, to construct the system matrices. A frequency shift is applied (if specified) and the dynamic matrix is calculated. To speed

up the process that determines the inverse of this matrix, a LU decomposition is performed first. The highest eigenvalues of the inverted dynamic matrix are calculated with a MATLAB function called `eigs`. These eigenvalues are inverted and shifted back to produce the lowest eigenvalues of the original system. The function determines next which of the calculated eigenfrequency is closest to the estimated eigenfrequencies. The resulting eigenfrequency is examined to determine if the solution is converged, another iteration step is necessary or if the iteration should be aborted. When a new iteration step should be performed, the calculated eigenfrequency is taken as input and the whole procedure is carried out again.

In a way, the iterative solver is a wrapper function that solves the system several times in an iterative process, each time with slightly different system matrices. The actual solving of the system is done by an internal MATLAB function called `eigs`. The arguments and options parsed to that function are controlled by the wrapper function.

## 11.2 Working of the solver

The first input argument is the name of the database file that holds the model data produced with B2IP. The second input argument is a vector that holds the estimates of the eigenfrequencies that the solver will try to calculate. Input argument three is optional and indicates either the thickness of the viscous layer or a MATLAB struct that holds settings of the function options. When the third argument is not specified the default function options are taken and the thickness of the layer is as specified on the database. If the function is called with four input arguments the third should hold the thickness of the viscous layer and the fourth an option structure.

When no output arguments are parsed to the function, it will display the determined eigenfrequencies. In case one output argument is parsed, the function will fill the variable with the eigenfrequencies. If a second argument is parsed, it will be filled with the corresponding mode shapes. In case a third argument is offered, the function will return the relative errors at the last iteration step in it.

### 11.2.1 Option structure

The option structure is a MATLAB struct that contains the settings of a number of options that influence the working of the solver. The different

option values are adopted from the structure or are set to a default value if no option structure was parsed. The different options and their default settings, are listed below:

- **tolerance** (tol): This is the relative deviation of the calculated eigenfrequency with respect to the frequency calculated in the previous iteration step. It indicates when the solution is considered converged. Default setting: $10^{-2}$
- **maximum iterations** (maxiter): When the solution does not converge in the number of steps indicated by this parameter, the function aborts the iteration process. Default setting: $10$
- **maximum difference** (maxdif): A third criteria to abort the iteration process. The parameter indicates the maximum allowed difference between the calculated frequency and the assumed frequency. (This parameter can speed up the process, since it is not necessary to perform the maximum number of iterations before abortion.) Default setting: $20$
- **frequency shift** (shift): In case the system that is to be solved includes a rigid body mode, the solver will always converge to the same frequencies or will not converge at all. The other modes of such a system can still be calculated when a so called frequency shift is applied to the system (See [2], page 247). The shift parameter indicates the size of the applied shift. Default setting: $0$ (no shift)
- **maximum frequencies** (maxfreq): In each iteration step, the lowest eigenfrequencies of the system are calculated. The parameter maxfreq determines how many eigenfrequencies are calculated in each step. (It goes without mentioning that if the number of calculated eigenfrequencies is chosen too small, the desired eigenfrequency may be not among them, and the solution will not be able to converge.) Default setting: $10$

The iterative solver uses a MATLAB solver to calculate the eigenvalues. The default tolerance settings of this solver is the MATLAB constant eps. The constant is defined as the distance from 1.0 to the next largest floating-point number. When a smaller tolerance level is parsed to the iterative solver it will automatically overwrite the default tolerance settings of the MATLAB solver.

The default setting for the maximum number of iterations is 300 for the MATLAB solver and is left unchanged.

The maximum number of modes (and frequencies) calculated by the MATLAB solver, is not the number of eigenmodes that the function uses internally to determine the solution. The function uses subspace iteration and the solution often converges faster when more modes than requested are calculated. More modes take more computing time per iteration step on the other hand. The MATLAB solver determines the optimum itself.

# Bibliography


[1] BELTMAN, W. *Viscothermal wave propagation including acouto-elastic interaction*. PhD thesis, University Twente, Enschede, The Netherlands, 1998.

[2] CLOUGH, R., AND PENZIEN, J. *Dynamics of structures*. McGraw-Hill Kogakusha, 1975. international students edition.

[3] THE MATHWORKS, INC. *Matlab reference manual / Internal matlab help*. 3 Apple Hill Drive, Natrick, MA 01760-2098, 2001.

[4] MERAZZI, S., AND STEHLIN, P. B2000 data structure and programming handbook. Tech. rep., SMR Engineering & Development, Bienne, Switzerland, 1994.

[5] MERAZZI, S., AND STEHLIN, P. MEM-COM reference manual. Tech. rep., SMR Engineering & Development, Bienne, Switzerland, 1994.

[6] MERAZZI, S., STEHLIN, P., AND DE BOER, A. B2000 processors reference manual. Tech. rep., SMR Engineering & Development, Bienne, Switzerland, 1994.

# A

# Defining the mex-functions

Each section in this chapter describes a group of function of the newly developed MEMCOM - MATLAB - API. The first section is concerned with opening and closing a MEMCOM session and with file handling. The second section deals with the functions involved in data transfer between MEMCOM and MATLAB. The functions that are involved in database management are described in the third section. In the last section of this appendix the functions in the function library that was developed are commented on.

The layout of the description of each function is identical and begins with a single line describing the functions purpose. Thereafter, the working of the function is commented on in more detail. If the function is overloaded the different tasks it can carry out are also described shortly. In some cases a short explanation of why certain tasks are grouped together to for one function is given.

The next part of the function description deals with the layout and implementation of the function. It is clear that not every line of code is commented on, but rather the structure of the code and the general approach to carry out a task or tackle a problem are described.

As a conclusion of each function description, the input and output arguments the function requires are listed.

## A.1 Opening and closing MEMCOM

### mcDBopenFile

Start up a MEMCOM session and open a database.

**Working** As with the C-API, MEMCOM is launched and a file is opened with a single function. Depending on the input arguments the function will only open old files, only create new files or create a new file when no old file exist. If only the filename is parsed as an argument, the function will only open an existing file or exit.
**Function layout** This function performs a relatively minor task, so the complete code is written inside the main function body. The first argument, a MATLAB character mxArray, is converted to a character string with the MATLAB function `mxArrayToString`. To extract the integer value of the second input argument, the function `getIntFromInput` is used. Lastly, the function `Int2mxArray` is called to put the integer value of the handle in the Output mxArray. Both functions are available in the function library and are commented on in section A.4.
**Input arguments** The function takes one or two input arguments. The first argument is a character string that holds the name of the file to be opened or created. The second argument is a flag that tells the function if it should open an old file, create a new file, or either open an old file or create a new file.
**Output arguments** On success, the function returns a handle (an integer value greater than zero) to the opened database.

## mcDBcloseFile

Close one or all databases or shut down MEMCOM.

**Working** This function is overloaded and can perform two tasks depending on the number of input arguments. In case no input argument is parsed to this mex function, it will save and close all open databases and close down MEMCOM. If a handle of a database is parsed to the function, it will close the corresponding database and return. Notice that, the function will not keep track of how many databases are still open. A MEMCOM session will continue to run even if all files are closed this way. This means that this mex-function has to be called once without arguments to close down MEMCOM [1].
**Function layout** The function performs one of two tasks. The first task

[1] It is recommended to call this function before MATLAB quits to make sure the MEMCOM session shuts down properly and all memory is freed. When MATLAB quits, it runs a script called finish.m, if it exists and is on the MATLAB search path. This is a file that you create yourself in order to have MATLAB perform any final tasks just prior to terminating. With this special feature it is possible to make sure `mcDBcloseFile` is always called once, and all MEMCOM sessions are closed before MATLAB terminates.

is validating a database handle and afterwards save and close the corresponding database. Since this is a relatively small task, the code is written in the main function body. The other task is captured in a subroutine which saves and closes all MEMCOM databases and afterwards closes down MEMCOM. The database handle, which can be parsed to the function as an argument, is retrieved and validated with the MATLIB library function `getDBHandleFromInput`.
**Input arguments** The function can take one input argument. Namely, the database handle of a database that is to be closed.
**Output arguments** No output arguments are returned by the function.

## A.2 Data transfer to and from MEMCOM

The following functions are involved in transferring data between MEMCOM and MATLAB.

### mcDBgetSet

Load a MEMCOM dataset into MATLAB.

**Working** In the C-API, retrieval of a dataset is not performed by the same function as retrieval of a relational table. The C-API also cannot retrieve more datasets at once (which is required when retrieving a skyline or sparse matrix). Still, the concept behind all actions described above is similar, namely the retrieval of datasets. It makes sense to apply function overloading to be able to make use of the same syntax for these actions.

The function is thus overloaded and performs a variety of (similar) tasks. Depending on the input arguments, the function can retrieve a relational table from a MEMCOM database and convert it to a MATLAB struct, retrieve a dataset from a MEMCOM database and convert it to a MATLAB numerical matrix, or retrieve multiple datasets from a MEMCOM database which form a skyline or sparse matrix, construct that matrix and put it in the MATLAB environment.

When the function is called with only a database handle and the name of the required set, the function assumes the dataset is a relational table or a full matrix. When the input type-flag is parsed, the function will attempt to open either a full, skyline or sparse matrix. In case an integer is parsed as a fourth argument, the function appends this number to the dataset name. This last feature makes it easier to address the different

cycles of sets which share the same set name (see subsection 5.3.2). If for some reason the retrieval of the data fails, the function will issue an error message and the command is returned to the MATLAB prompt.
**Function layout** Since this mex-function is overloaded multiple times, all operations the function performs are coded in separate subroutines. The function makes extensive use of the MATLIB function library. First the validity of the input arguments are checked and converted to data types supported in C. Depending on the number and values of the input arguments, the function than calls one of the following subroutines.

- `mcDBgetSet_RTABLE` extracts a relational table from a MEMCOM database and stores its data in a MATLAB struct. The routine uses the function `getRTableContent`, stored in the MATLIB library, which performs the actual extraction of the data and returns the output mxArray.
- `mcDBgetSet_FULL` retrieves a full matrix from a MEMCOM database and creates a MATLAB numerical matrix with the data.
- `mcDBgetSet_SKYLINE` loads the two datasets that make up a skyline matrix from a MEMCOM database and construct a numerical matrix from them in MATLAB. The routine uses the MATLIB library function `AssembleFULLfromSKYLINE` that performs the actual reconstruction of the matrix after the datasets were retrieved.
- `mcDBgetSet_SPARSE` is the equivalent of `mcDBgetSet_SKYLINE` for sparse matrices. Since MATLAB supports sparse matrices, there is no need to construct a full matrix from the dataset. Instead the pointers to the datasets can simply be assigned to the corresponding pointers of the mxArray. In case the data is complex, the MATLIB library function `mcUnZipRI` is used to 'unzip' the real and imaginary part of the complex numbers from the continuous data stream.

After one of the above routines is carried out, the function transposes the constructed matrix. This is necessary because the storage of data is different for MATLAB and MEMCOM. MATLAB stores all its data column wise, MEMCOM stores it in a row wise fashion.
**Input arguments** The function takes two to seven arguments. The first argument is the database handle. The second argument is the name of the requested set. The third argument is the type of the requested numerical matrix. Argument four is an integer indicating the cycle in which the dataset was created or used. The fifth and sixth arguments indicate the

size of the matrix. The last argument determines if the matrix elements contain real or complex data.
**Output arguments** The function returns an mxArray containing the requested dataset.

## mcDBgetDesc

Load a descriptor of a dataset into MATLAB.

**Working** This function performs a similar task as the `mcDBgetDesc` function in the C-API. The function is not overloaded and simply retrieves a descriptor from its corresponding dataset on a MEMCOM database and converts it to a MATLAB struct.
**Function layout** Despite the fact that this function is not overloaded, the code that performs the action is captured in a separate routine. The validity of the handle and existence of the requested descriptor on the database are verified before the attempt to retrieve any data is made. Since a descriptor is a special case of a relational table, the code that converts a relational table to a MATLAB struct used in the `mcDBgetSet` function can be reused for this function.
**Input arguments** The function takes two input arguments. The first argument is the database handle and the second argument is the name of the set, with which the descriptor corresponds.
**Output arguments** One output argument is returned, namely a MATLAB struct containing the descriptor data.

## mcDBputSet

Write a MATLAB variable to a MEMCOM dataset.

**Working** The same design decisions that apply to the `mcDBgetSet` function apply here. In a sense, this is the inverse function of `mcDBgetSet` and it is good programming practice to reflect this in the way the function is called and operates. Thus, this function is also overloaded and performs the inverse of the tasks the `mcDBgetSet` function performs. It can convert a MATLAB struct to a relational table and write it to a MEMCOM database, convert a MATLAB numerical matrix to a dataset and write it to a MEMCOM database or convert a MATLAB numerical matrix to skyline or sparse form and write the corresponding datasets to a MEMCOM database.

When the function is called with only the database handle, set name and name of the MATLAB array which holds the data, the function assumes the output type is either a full matrix or a relational table. If the output-type-flag is raised the function will write a numerical matrix to the database as a full, skyline or sparse matrix. In case a integer is parsed as the fourth argument, the function will append this number to the dataset name. This feature is included to make addressing datasets which are part of a cycle easier (as with `mcDBgetSet`).

If the dataset does not exists on the database, it is created. In case the set does exist, but is not large enough to hold the new data, the set is deleted from the database and a new set is created. When the dataset exist and is large enough to hold the new data, the old data simply gets overwritten and the surplus of memory is set to zero. (Notice that, when such a set is read again into MATLAB the old size of the set is still valid and all the zero's will be loaded into MATLAB as well. To avoid this, the old dataset can be manually removed from the database with `mcDBdeleteSet`. Frequent use of this technique will result in a Swiss-cheese-database however.) If for some reason the retrieval of the data fails, the function will issue an error message and the command is returned to the MATLAB prompt.

**Function layout** Since this function is the inverse of the `mcDBgetSet` function, it is not surprising that the layouts of both functions strongly resemble each other. The function also uses four separate routines to perform each task, it uses `mcZipRI` to zip the real and imaginary parts of complex numbers and it transposes the matrices before they are written to a MEMCOM database to account for the difference in data storage between MEMCOM and MATLAB.

**Input arguments** The function takes three to five input arguments. The first argument is an mxArray that holds the data that is to be written to the MEMCOM database. The second argument is a database handle. Argument three is the name that is used to indicate the set on the MEMCOM database. The fourth argument indicates in what form (full, skyline or sparse) the numerical matrix should be written to the database. The last argument indicates the cycle in which the dataset is created or used.

**Output arguments** No output arguments are returned by this function.

## mcDBputDesc

Write a MATLAB struct to a MEMCOM descriptor.

**Working** As `mcDBputSet` is the inverse of `mcDBgetSet`, `mcDBputDesc`

is the inverse of `mcDBgetDesc`. The function is not overloaded and writes a descriptor to a dataset on a MEMCOM database after converting it from a MATLAB struct.

**Function layout** The inverse of `mcDBgetDesc` is also not overloaded and the code that performs the actual task is captured in a separate routine. The function checks if the parsed arguments are a valid database handle, an existing dataset and a MATLAB struct. If this is not the case, the function returns an error message. A part of the code, used in `mcDBputSet` to convert a MATLAB struct to a relational table, can be reused in this function.

**Input arguments** Three input arguments are required in the call to this function. The first argument is a MATLAB struct that holds to data of the descriptor. The second input argument is a database handle. The last argument is the name of the set to which the descriptor is to be attached.

**Output arguments** No output arguments are returned by this function.

## A.3 Database management

The functions described below are intended to manage the database from within MATLAB and inquire after its properties.

### mcDBinqSetAtt

Returns a MATLAB struct containing attributes of a set.

**Working** This function is overloaded and strongly resembles the two C-API functions `mcDBinqSetAtt` and `mcDBgetSetNextIter`. Both functions inquire after the properties of a dataset on a MEMCOM database.

If this function is called with a database handle and a set name as arguments, the function returns a MATLAB struct containing the set properties obtained with the C-API function `mcDBinqSetAtt`. If the requested dataset does not exist, the function returns zero. The function can thus be used to verify the existence of a dataset before an attempt to retrieve it is made.

If the function is called with a database handle and an iterator, the function returns a MATLAB struct containing the set properties of the set pointed to by the iterator and the iterator value that points to the next set. The function can thus be used to iterate through the database while obtaining the properties of each set. When the iterator parsed to the function

is set to zero, the properties of the first set are retrieved. When the last set it reached the function will return zero on the next function call.
**Function layout** This mex-function is overloaded, but since the implementation of both task are relatively small, they are both written in the main function body. The validity of the database handle parsed to the function is checked before an attempt to retrieve any data is made. In both tasks, the subroutine `getSetAttributes` is called that converts the B2000 data structure `mcSetAttributes` to a MATLAB struct. The function `intArray2mxArray` converts an array of type `int` to an mxArray. (One of the fields of the `mcSetAttributes struct` is an array of type int)
**Input arguments** The function takes two input arguments. The first argument is a database handle. The second argument can be either a name of a dataset or an iterator.
**Output arguments** Depending on the input arguments, one or two output arguments are returned. If a name of a dataset is parsed as input argument, the function returns a MATLAB struct containing the set properties or zero. In case an iterator is parsed to the function, it returns a MATLAB struct containing the set properties and the new value of the iterator.

## mcDBdeleteSet

Remove a set from a MEMCOM database.

**Working** When a dataset is no longer needed and its set name is needed again on the same database, the unused dataset can be manually 'removed' from the database with this function. Since the function does not really remove the data on the database and the disk space can not be reclaimed, it is advisable to use the function scarcely.
**Function layout** The implementation of this functions is straightforward. The existence of the database and dataset are checked and then the dataset is removed. Despite the simple tasks, the code is captured in a separate routine.
**Input arguments** Two input arguments are required in the call to this function. The first is a database handle and the second is the name of the set that is to be removed.
**Output arguments** No arguments are returned by this function.

## mcDBdeleteDesc

Remove a descriptor from a MEMCOM database.

**Working** This function is the equivalent of the `mcDBdeleteSet` function and removes a descriptor of a dataset from a database. A difference with the `mcDBdeleteSet` function, is that this function can be used without hesitation since the disk space reserved for a descriptor is reclaimed after it is removed from the database. (A descriptor can be deleted and added again without the need for extra disk space.) Notice that a function that removes the descriptor is not available in the MEMCOM - API at present so the function is inoperable and only an error message will be returned[2].
**Function layout** The implementation of this function resembles that of the `mcDBdeleteSet` function. The existence of the database and dataset are checked and thereafter, the descriptor is removed. The code is captured in a separate routine.
**Input arguments** The function takes two input arguments. The first is a database handle and the second is the name of the set that corresponds to the descriptor that is to be removed.
**Output arguments** No arguments are returned by this function.

## mcDBpack

Defragment a MEMCOM database.

**Working** Because the `mcDBdeleteSet` function does not free the disk space occupied by the deleted set, a defragmentation function is available. (The term 'defragmentation' is somewhat misleading because the data still in use is not scattered.) The function puts all remaining data in one contiguous block of disk space so the unused disk space is freed.
**Function layout** The code of this function consists, apart from the code to handle the input argument, of not more than one function call to `mcDBpack` and one error message.
**input arguments** The function takes only the database handle that is to

[2]The function call to MEMCOM that performs the actual work is commented out. When the MEMCOM - API function to remove a descriptor is added, this function can be activated with relative ease by removing the comment. Hereafter the function has to be recompiled in MATLAB by entering `mex mcDBdeleteSet` in the command line while the active search path is the directory that holds the function's code.

be 'packed' as input argument.
**Output arguments** No arguments are returned by this function.

## A.4 B2MATLIB function library

This section consists of a short description of all functions available in the MATLIB function library. The working and structure of the different functions are outlined rather than a complete description of design decisions made during implementation.

### getDBHandleFromInput

This library function extracts the value of the database handle after checking the type of the argument. If all is well, the function `mcDBisHandle`, available in the C-API, is used to validate if the handle points to an opened database. The function returns the handle or produces an error message and aborts the calling mex-function.

### getValueType

This function is used to check the data type of numerical matrices. The function returns a flag that indicates if the data is of type `int` or of type `double` or it issues an error message that was parsed to the function.

### checkDataSetOnDB

This function is overloaded and can perform two functions. If the function is called with a database handle, set name and the required set size, it first checks if the dataset exists on the database. If the set does not exists, it is created and the function returns true. In case the set does exist and its size is equal or larger than the required size the function returns true. In case the set does exist but its size is smaller than the required size, the function deletes the old set, creates a new one with the required size and returns true.

If the function is called with the last argument equal to zero, it will return true on existence of the set and false on non-existence.

### Int2mxArray and Double2mxArray

These functions take an instance of either `int` or `double` as input argument, create a mxArray of the appropriate type, insert the data and return the pointer to the new mxArray.

### getIntFromInput

All mex-functions taking an integer as input use this function to convert the input mxArray to a standard integer. Since MATLAB considers all numerical data to be of type `double`, unless the user explicitly defines an other type, the function most be able to convert both mxArrays of type `int` and `double`.

### mcZipRI

This function combines the real and imaginary data from mxArray like a zipper, fills a newly allocated part of memory with it and returns the pointer to the allocated memory. The function is split up into two similar blocks. One block that handles data of type `int` and one block that handles data of type `double`.

### mcUnZipRI

This is the inverse of the `mcZipRI` function. It 'unzips' the complex data from a contiguous data string and fills a MATLAB array that was parsed to the function with the real and imaginary part. This function is also split up into a block that handles data of type `int` and a block that handles data of type `double`.

### AssembleTYPEfromTYPE

The following functions are described simultaneously in this subsection:

- `AssembleSKYLINEfromSPARSE`
- `AssembleSKYLINEfromFULL`
- `AssembleFULLfromSKYLINE`
- `AssembleSPARSEfromSKYLINE` (Not implemented)

These functions perform the actual work in the transformation between the different matrix storage types. The tasks that are carried out by these functions are constructing a matrix from its corresponding datasets and deriving the dataset that make up a sparse or skyline matrix. Further details of their implementation are not commented on here, since the structure of their code is straightforward (and not particularly insightful). The function `AssembleSPARSEfromSKYLINE` was not implemented[3]. The choice was made to use the `AssembleFULLfromSKYLINE` function instead, and afterwards convert the resulting matrix to sparse format in MATLAB.

## getFieldContent

The functions `mcDBputSet` and `mcDBputDesc` use this function to extract the data from a mxArray of type `struct`. The function iterates through the fields of the `struct` and creates for each field an mxArray to which the data of that field is parsed. All created mxArrays are assigned to an array and the function returns a pointer to this array.

## getRTableContent

This function, used by `mcDBgetSet` and `mcDBgetDesc`, iterates through the fields of a relational table that was parsed to it. An mxArray is created and the fields of the relational table are added to it as fields of a `struct`. When all data in the table is converted, the function returns the pointer to the created mxArray.

[3]The conversion from a sparse matrix to a full matrix requires sorting of the vectors which describe the different columns in a sparse matrix (see subsection 9.2.2). Since mex-functions are written in C and not C++, the class `vector` used in the implementation of B2AEMSAC, which offers sorting possibilities, is not available.

# B

# B2MATLAB

The coupling between MEMCOM and MATLAB as described in chapter 3 to 7 is designed to work from within MATLAB. The coupling always requires that a MATLAB session is started up and all other processors can only be called from the MATLAB command line. This can work somewhat limiting in the sense that it is not possible to call MATLAB routines from other processors. The coupling is so to speak one-way only. The processor and corresponding control module described in this appendix offers the possibility to use the coupling the other way around.

## B.1 Working

The control module of B2MATLAB is called with a pointer to a character string containing the name of the MATLAB script that is to be run. If the pointer is not set to NULL, the MATLAB engine is started up and the MATLAB command or -script in the parsed character string is executed in MATLAB. If an error should occur while executing the MATLAB command or -script the control module displays an error message followed by the MATLAB output and returns the value $-1$. In case no errors occur, the MATLAB output is displayed and the control module returns zero after finishing the script.

If the pointer parsed to the control module is set to NULL, the control module will start an interactive MATLAB session. The standard input and output will be coupled with the input and output of the MATLAB engine. Multiple calls[1] to the engine can be made from the shell prompt and the results will be displayed in the shell. The interactive session can be termi-

[1] If multiple commands have to be executed in MATLAB without interference of the user, it is of course advisable to capture the commands in a MATLAB script and call the B2MATLAB processor with the file name of the script as argument.

nated by entering the command 'quit'.

## B.2 Processor layout

B2MATLAB consists, like all B2000 processors, of a main function and a control module. The main function fulfills a minor job, namely obtaining the name of the MATLAB script to be executed and calling the control module with this filename as argument.

The control module will start up the MATLAB engine and afterwards will call one of two functions depending on the input argument parsed. The execution of a MATLAB script is captured in a separate function. The second function contains the code involved in the interactive MATLAB session. Both functions return zero on success and minus one on failure. The control module will print an error message if an error is returned by one of the above functions and will return $-1$ itself.